Reuse

Chapter 18 of Using UML is the main source of material for this section: read this now. (There is a little more overlap than usual between these notes and the book, but you still need to read both.)

The topic of reuse actually ties together many of the considerations of the final part of SEOC2: quality, quality assurance, configuration management, verification, validation, testing, management, leadership and teamwork are all relevant to a situation in which an organisation seeks to increase the level of reuse it gets.

To illustrate this we take the following hypothetical situation (actually it's not all that hypothetical, it's an adaptation of the situation of an organisation I once did some work for).

How to achieve reuse?

Your client is fairly mature in software development (CMM level 4, maybe). It is a service provider, and the provision of its service requires a high level of software support. That is, its core business is "IT intensive" although software development itself is not the organisation's core business.

The software developers are distributed throughout the UK in around 10 software development centres, each of which draws its customers from the rest of the organisation. The sites compete for customers.

The client has made several previous attempts at instituting a reuse programme, with limited success. Significant cross-site reuse has always been rare.

Slide 69

Analysing the problems

1. What can be reused?
2. Why is it hard to achieve high levels of reuse?
3. What are the dangers of ill-thought-out reuse?
4. What are the characteristics of genuinely reusable components?
5. When can they do the extra work that making reusable components involves?
6. What difference does object orientation make?
7. What are the motivations for achieving high levels of reuse?
8. What are the interactions between quality culture and reuse?

Slide 70

What can we reuse?

- Most obviously, code: classes, functions
- Designs, even sections of analysis: remember design patterns
- Documentation, e.g. standard bits of user manuals?
- Tests, e.g. what happens if we run out of memory/disk
- (expertise, tools – may or may not count these as reuse)

Slide 71

What are the dangers of ill-thought-out reuse?

- It may not work – and if you didn't write it, it's easy for it to be harder to debug it than it would have been to write it in the first place
- It may not do exactly what you want, and you may end up twisting the requirements to fit what's available (but beware: this is the classic excuse for never doing any reuse at all!)
- You may end up with many slightly incompatible versions of the same thing, which can be horribly confusing (big CM problem).
- It may work, but not be properly documented, and you may misunderstand it.
- You may have to import things you don't need along with the functionality you want to reuse, getting software bloat.

Slide 72

Why is it hard to achieve high levels of reuse?

- It's harder to reuse components the bigger they are
- Change may make "reusable" components outdated so they never get reused: any extra effort involved was wasted
- Avoid duplicating OS or library facilities: the easiest "reusable" things to build are useless because the functionality is already conveniently available (buying is almost always better than building)
- Finding the right thing, even when it's there! Once component libraries get bigger than a few hundred things the searching problem is very serious
- NIH syndrome – not invented here! (Justifiable? Whose head is on the block if it doesn't work?)

Slide 73

In this particular case, we can also observe that competition between sites is bound to affect cross-site reuse.

What about incentives for reuse? It's an obvious thing to try. (Side issue: incentives for adding things to a reuse library, or incentives for taking them out and using them? It may seem obvious, but it's rather important to make the latter at least more prominent than the former – "reusable" things are no use, only reused things...) In fact (according to Michael Wasmund's panel discussion abstract in the International Conference on Software Engineering 1994) IBM Böblingen tried both incentives for reuse and reuse targets. They found that although incentives helped to raise awareness of the issues, it was explicit targets for reuse that made the difference in what actually happened.

What are the characteristics of genuinely reusable components?

- architecturally compatible
- sufficiently general, in the right way!
- properly documented with specification
- thoroughly tested

Not only does it cost, but it's hard, and if you get it wrong the cost is wasted.

Slide 74

So there is some extra work to be done to make things reusable: e.g., a component has to be documented much more thoroughly if it is to be used later by people outside the project in which it was built, than if the documentation is only intended to be read by people who know a lot about the context and the current project.

When can we do the extra work that making reusable components involves?

Possibly the single main reason why reuse efforts fail is that the answer is "we can't". Here are some options (classification of Henderson-Sellars and Pant, Object World, 93):

- End of project: hard to find resources?
- G-C1 model: before delivery, so customer pays
- Two-library model: as time permits (will people remember?)
- Alternative core-technology model: permanent team to do reuse work (but are they the best people to do it, do they understand the components?)

Slide 75

What difference does object orientation make?

- OO encourages the high cohesion/low coupling style that is anyway good practice
- Object oriented analysis concentrates on problem domain objects, which are more stable than the functionality of a particular application.
- Aside from stability, problem domain objects are more likely to occur in several different contexts than functionality on them. A company can develop a collection of *business objects* reflecting the common entities in their own business.
- Inheritance allows developers to use the *Open-Closed* rule – develop classes that are both closed (usable now) and open (extensible/modifiable later). Maintenance done on a base class can automatically affect subclasses.
- Working in terms of objects makes missing but "obviously sensible" functionality more obvious.

Slide 76

One of the main claims for object orientation is that it enables higher levels of reuse than traditional software development methods. People do quote reuse percentages – sometimes as high as 70 percent – but beware, these are practically meaningless without a good definition of what's being measured!

What are the motivations for achieving high levels of reuse?

- saves money: don't need to pay to develop the reused component
- improved reliability, by cumulative debugging of reused components
- reduced market cycle time (quicker development)
- reduced effort (ditto)
- encourages modularity

Notice that though the most obvious saving is on development of the reused thing, the biggest saving seems to be due to higher quality, hence lower maintenance costs. (Martin Griss, HP)

Slide 77

The full reference is Martin Griss, Software Reuse Experience at Hewlett Packard, International Conference on Software Engineering, 1994.

What are the interactions between quality culture and reuse?

- Can reuse help us to achieve high quality software? Perhaps, by improving reliability and encouraging good modular design (see above)
- Can reuse prevent us from achieving high quality software? Perhaps, by encouraging us to bend the requirements to fit the available components.
- Can encouraging high quality (how?) enable reuse? High quality components are more likely to be reusable, and if things are routinely well documented and tested the gap between normal and "reusable" components may be less.
- Can encouraging high quality (how?) make reuse difficult? Well, a too-rigid quality assurance program can make reuse difficult to achieve, e.g. because a reused component may not have been built according to the current QA regs: is it OK to reuse it?

Slide 78